Chain Replicated Block Store

Ethan Brown

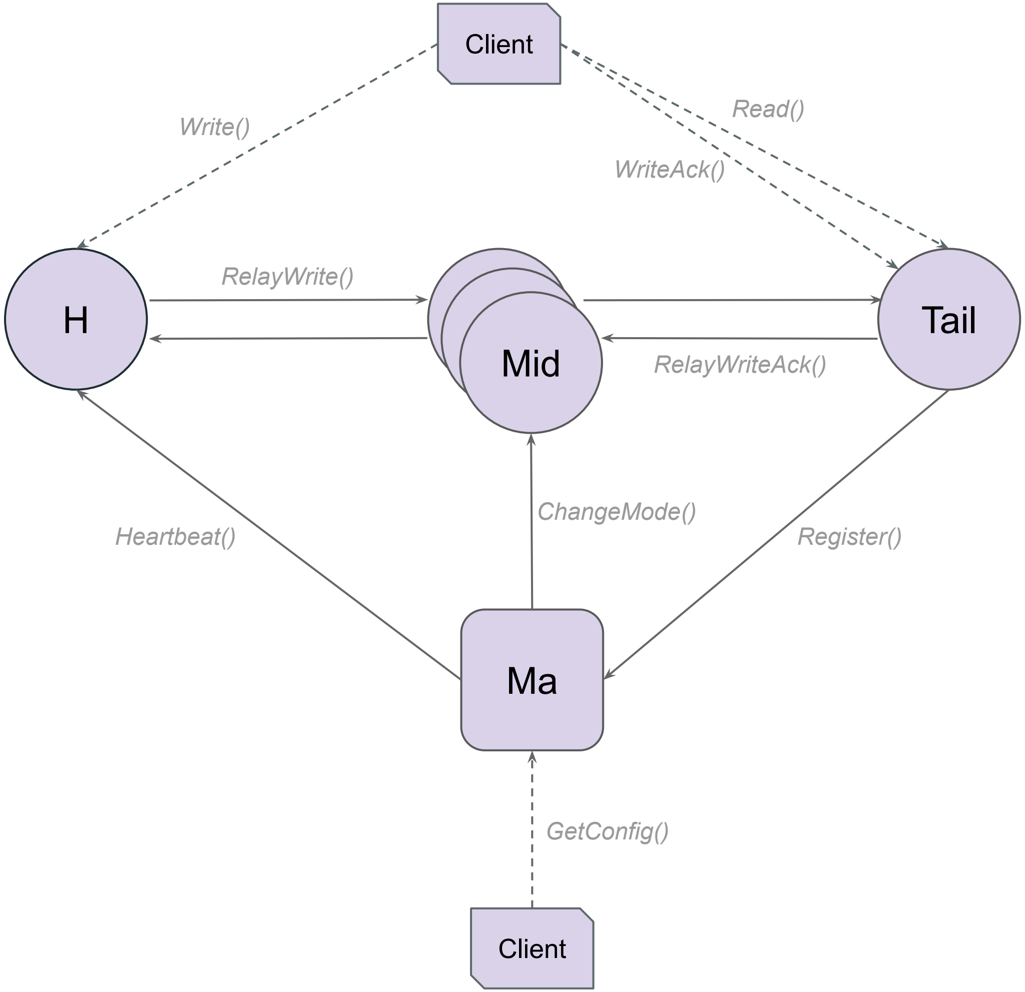
Marvin Tan

Sam Kottler

Todd Hayes

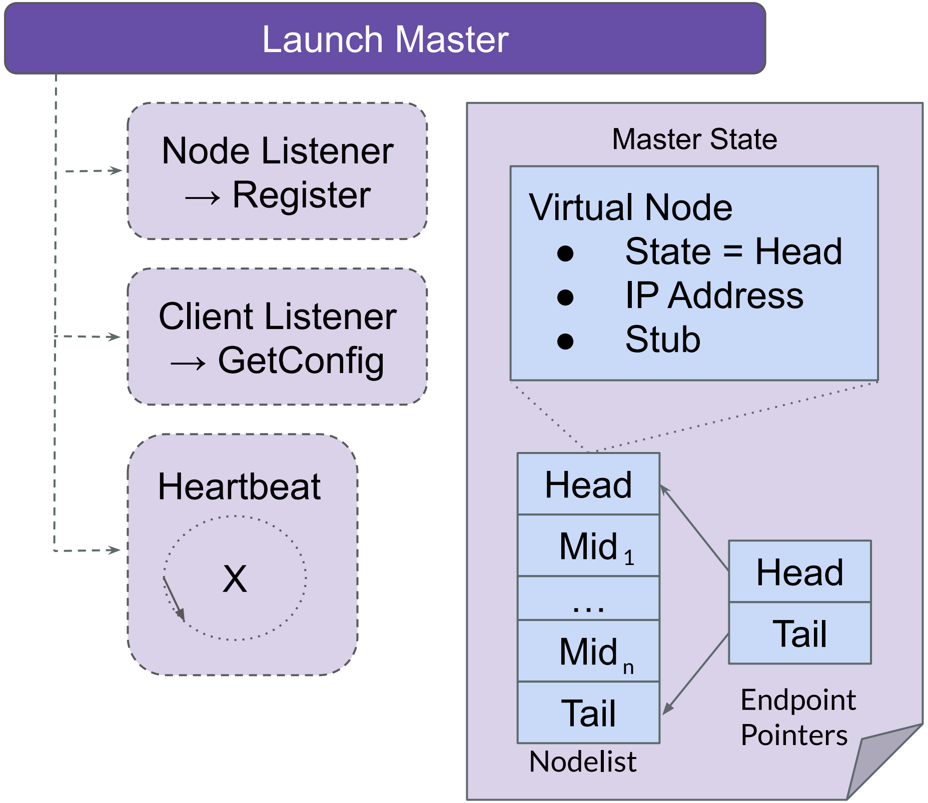
Architecture

- Multi-Node Chain Replication
  - 1 to N nodes in chain
  - Chain managed by Master
  - Client gets chain information from master, speaks with head and tail
- Consistency
  - Linearizable up to N-1 Failures
  - Durable and Consistent up to N Failures
- C\C++ with gRPC
- All benchmarks run on cloud nodes
  - Ubuntu 20.04
  - 4Gb Memory
  - 10 core 2.6GHz cpu
  - HDD storage

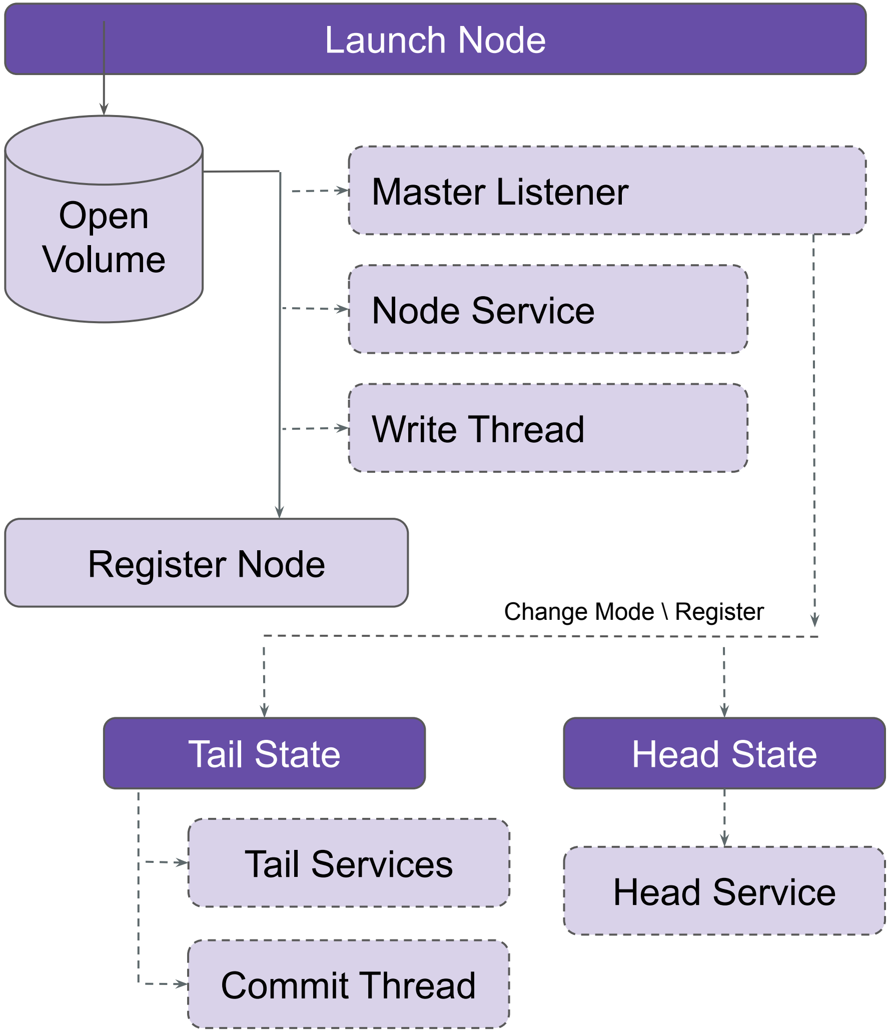


Architecture - Master

- Coordinates chain
  - Tracks nodes in chain
  - Handles membership
  - Informes nodes of state changes
- Communicates Endpoints to client



Architecture - Node



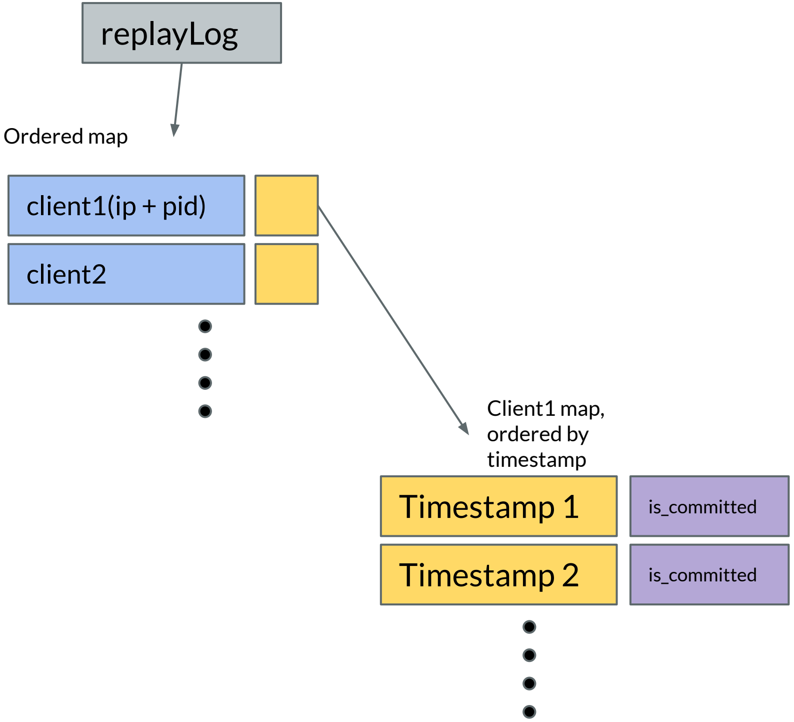
- Nodes run a variety of services that facilitate communication
- Master can alter state, which changes services exposed
- Blocks stored in a volume

Client Library

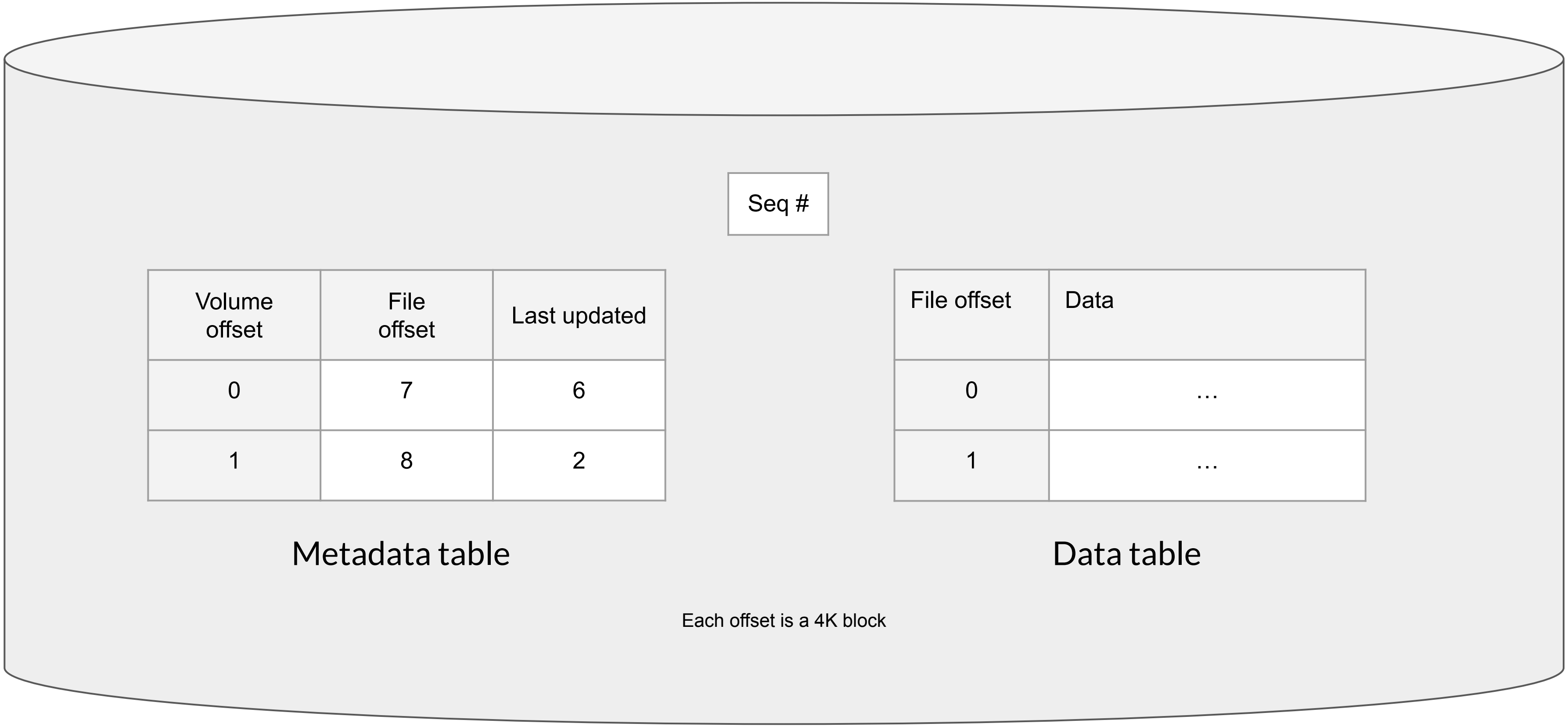
- Clients generate request IDs(clientIp: clientPid:timestamp) for each write
  - A pair of Client IP : Client PID is generated upon library class initialization
  - Timestamp is generated upon each write
- **Read** (&data, file_offset)
- **Write** (data, file_offset, &timestamp)
  - Generate the timestamp upon call
  - Return right away. Add the write to a list of pending writes
- **AckWrite**(timestamp)
  - Call tail to see if the write for a give timestamp is committed
  - This function also return as soon as tail gets the result. If the client wants to do things like retry after x seconds of write not acked, they will need to use their own timer.
- **RetryTopPendingWrite**(&timestamp)
  - Resend the request with a certain. This retry will only be sent if the timestamp is present in the list of pending writes
- **PopPendingWrite**(&timestamp, &pending_write_entry) and **PeekPendingWrite**(&timestamp, &pending_write_entry)
  - Both check top of the pending write list. Can be used to remove acked writes or skip writes that you don't want to acknowledge at all
- **RefreshConfig**():
  - Private method. Contacts the master and requests the address of the head and tail nodes. This is called whenever a request to a head or tail fails with connection error

Replay Log

- **AddToLog**(client_request_id)
  - Called by every node: Adds incoming client request IDs to replay log
  - Used to detect retries. Return appropriate value based on whether the request id is already present in the log
  - Log added as *non-committed*
- **CommitLogEntry**(client_request_id)
  - Called by any node when the request has been committed(indicate by RelayWriteAck grpc call passed upstream)
- **AckLogEntry**(client_request_id)
  - Tail node calls this when client calls in ack. Removes a committed entry, and any entry belong to the same client that is older(assume client will try to ack request in order)
  - Any change to replay log is passed upstream
- **CleanOldLogEntry**(age);
  - Replay log garbage collects stale Client request ids that is too old



Volume



Seq #

Volume offset	File offset	Last updated
0	7	6
1	8	2

Metadata table

File offset	Data
0	…
1	…

Data table

Each offset is a 4K block

Volume

Atomic commit

- Copy-on-write for data
- Copy-on-write for metadata
- Multi-level table for metadata
- Scales well with larger volumes

```
Write(data, vol_offset, seq_num):
  write(data, new_file_offsets);
  write(metadata, new_md_offsets);
  record_uncommitted(vol_offset, new_file_offsets);
  record_pending(seq_num, new_md_offsets);

Commit(seq_num, vol_offset):
  get_pending(seq_num);
  fsync();
  write(seq_num, new_md_tab, 0);
  fsync();
  add_old_blocks_to_free_list();
```

Non-atomic commit

- Copy-on-write for data
- Single level metadata table
- Faster metadata accesses
- May have partial writes if all nodes crash

```
Write(data, vol_offset, seq_num):
  write(data, new_file_offset);
  record_uncommitted(vol_offset, new_file_offset);

Commit(seq_num, vol_offset):
  get_uncommitted(seq_num);
  write_metadata(vol_offset, new_file_offset);
  fsync();
  write(seq_num, 0);
  fsync();
  add_old_blocks_to_free_list();
```

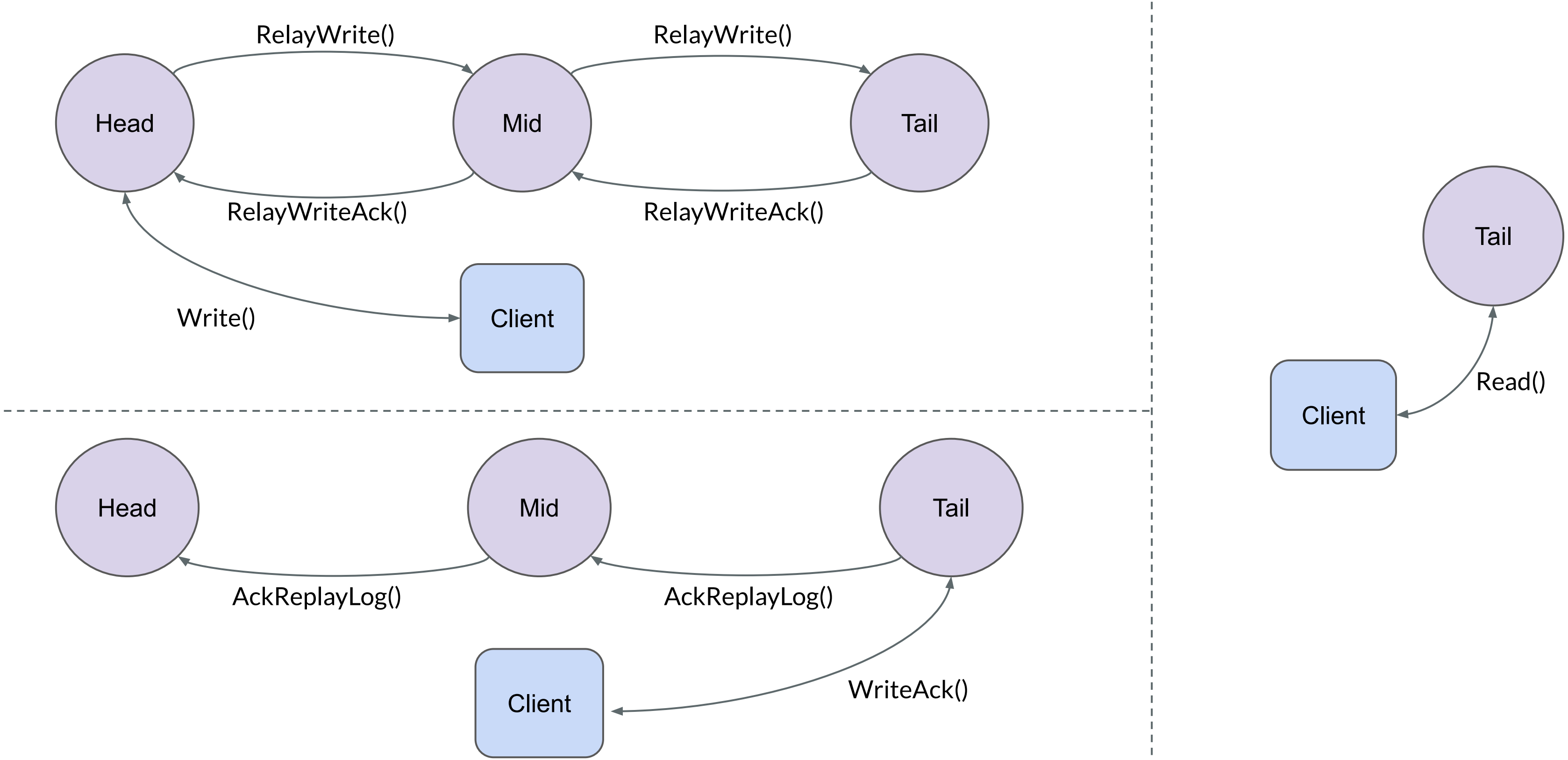
Volume

	Atomic	Non-atomic
Aligned write	19 us	4.6 us
Aligned read	3 us	2.3 us
Unaligned write	25.6 us	10.5 us
Unaligned read	5.9 us	4.1 us
Aligned commit	1.7 ms	2.5 ms
Unaligned commit	1.7 ms	5.8 ms
Write-commit	12 ms	12 ms

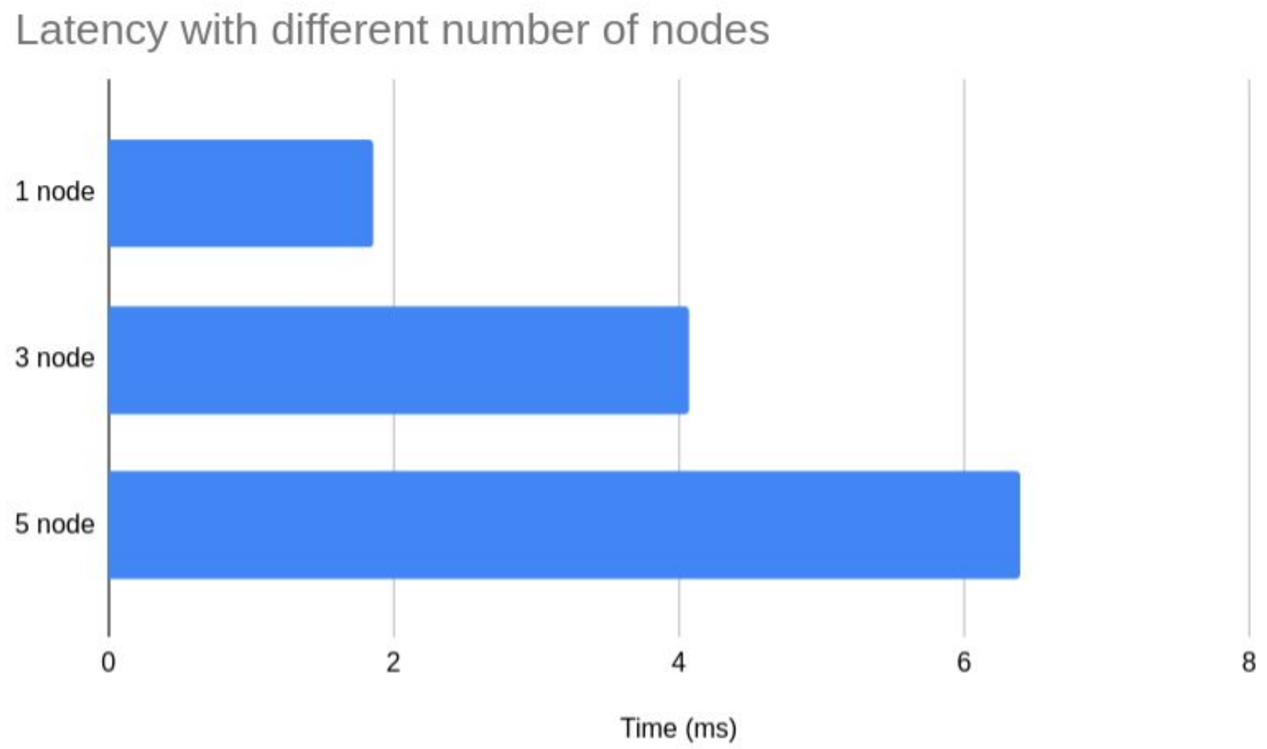
Message Propagation

- Head Service
  - Write() adds to pending list and replay log
- Node Service
  - Background ‘write thread’ pulls from pending list, writes, forwards, and adds to sent list
  - RelayWrite() adds to pending list and replay log
  - RelayWriteAck() removes from sent list, commits, and forwards
  - AckReplayLog() removes from replay log and forwards
- Tail Service
  - Background ‘commit thread’ pulls from sent list, commits, and forwards
  - Read() returns data
  - WriteAck() checks commit, removes from replay log, and forwards

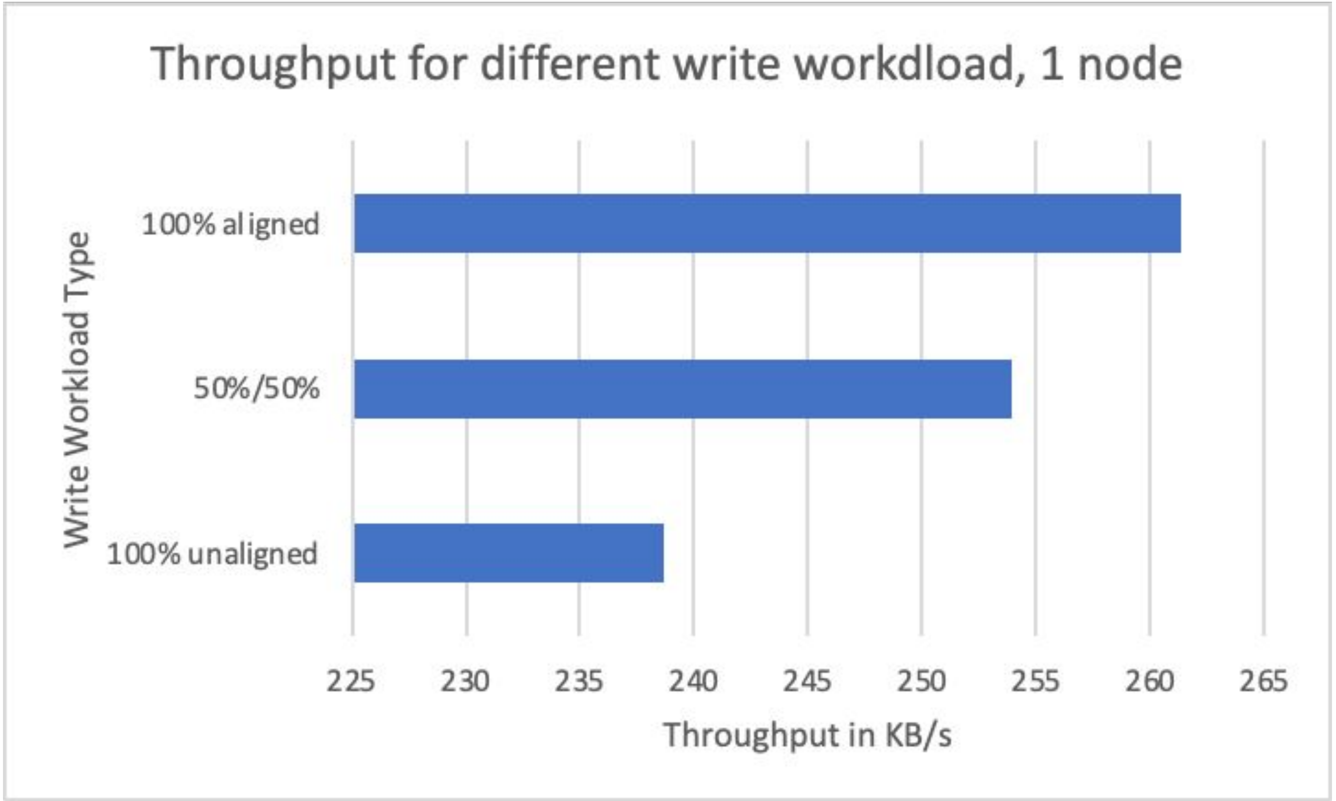
Message Propagation



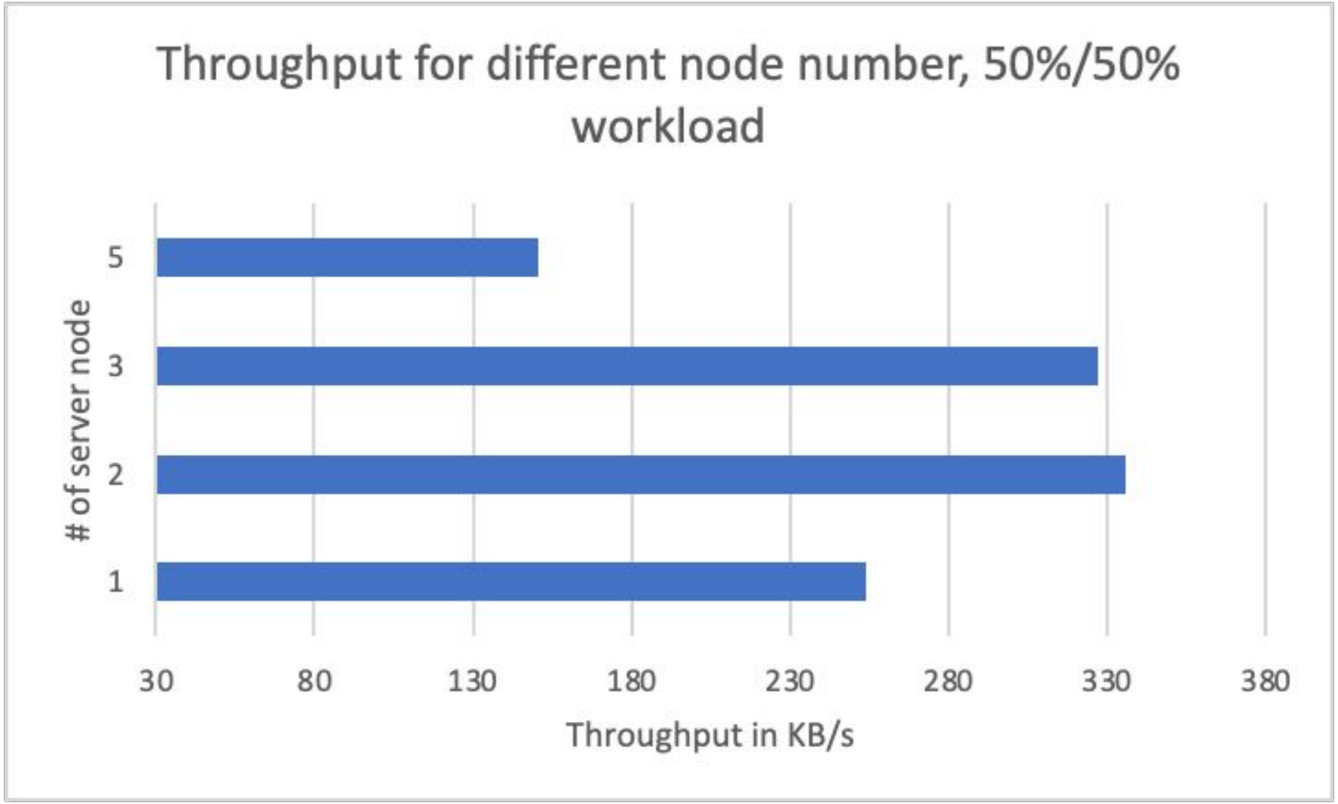
Latency benchmark - write



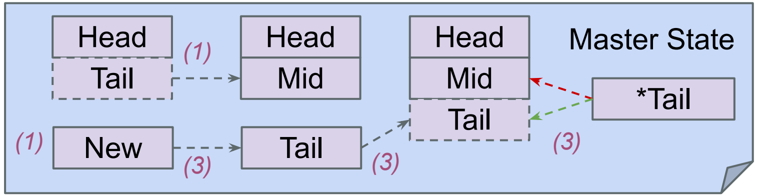
Throughput benchmark - write mode



Throughput benchmark - effect of node number



Registration & Restoration

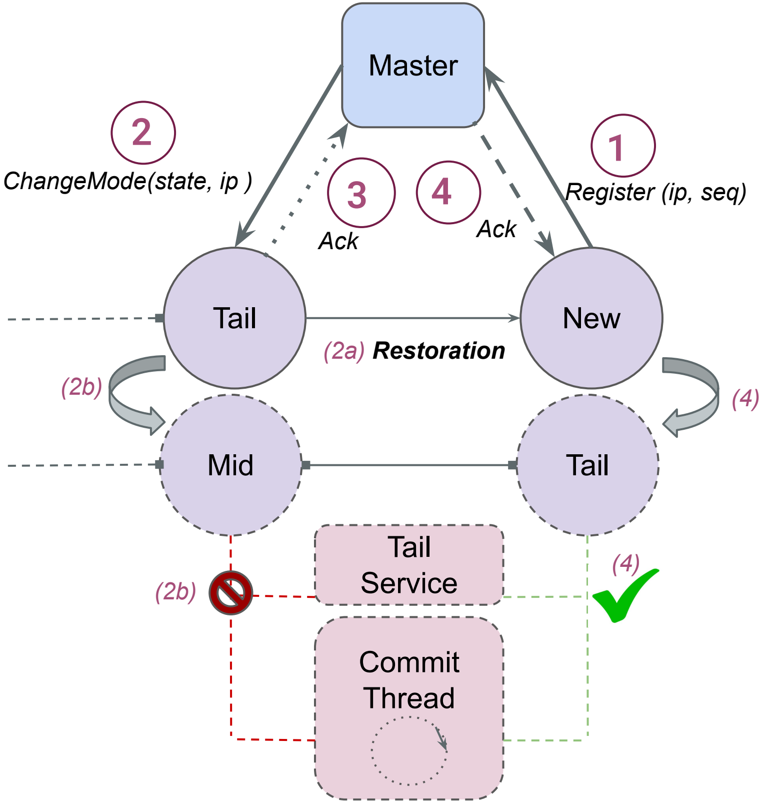


Existing Chain

1) New nodes register with master, master generates virtual node
2) Master notifies tail of new state, tail generates downstream channel
    a) Tail restores new node
    b) updates own state, kills tail service
3) Master adds new node to node list and updates tail pointer
4) New node updates internal state, launches tail service and commit thread

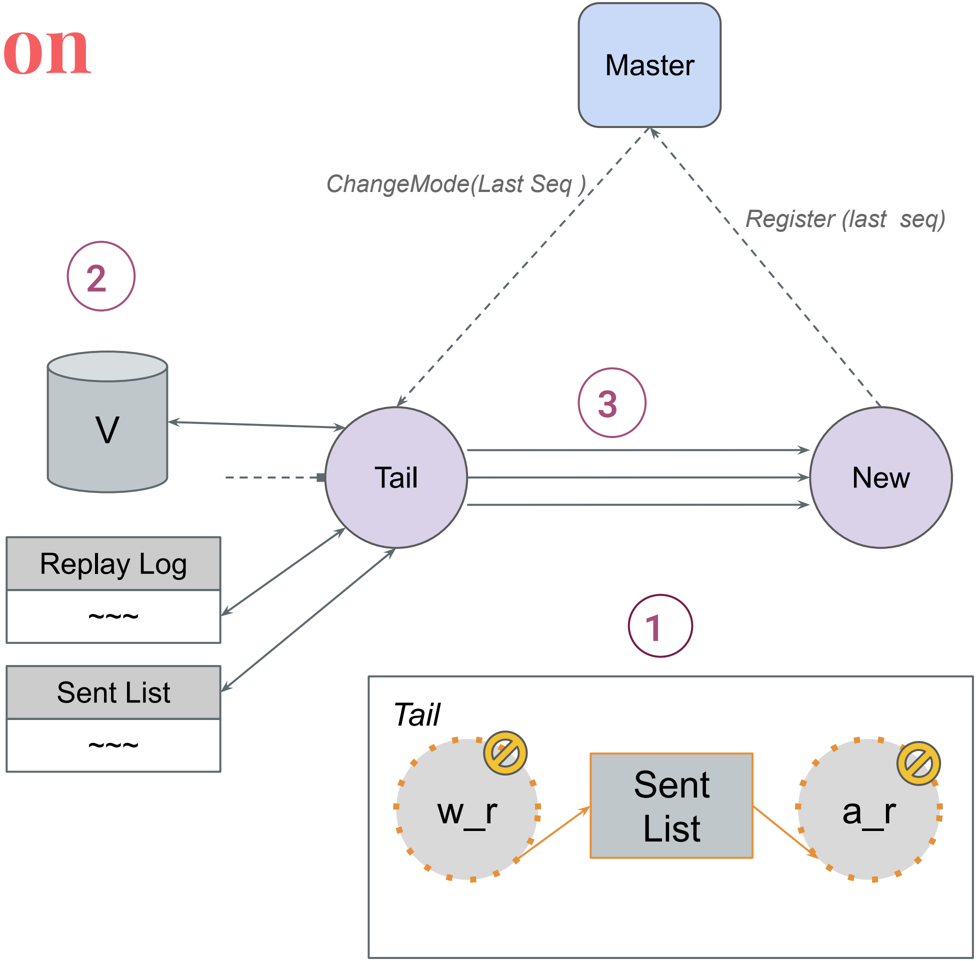
New Chain

- Skip step 2
- New node becomes Single Server
- Launches Head & Tail services \ commit thread

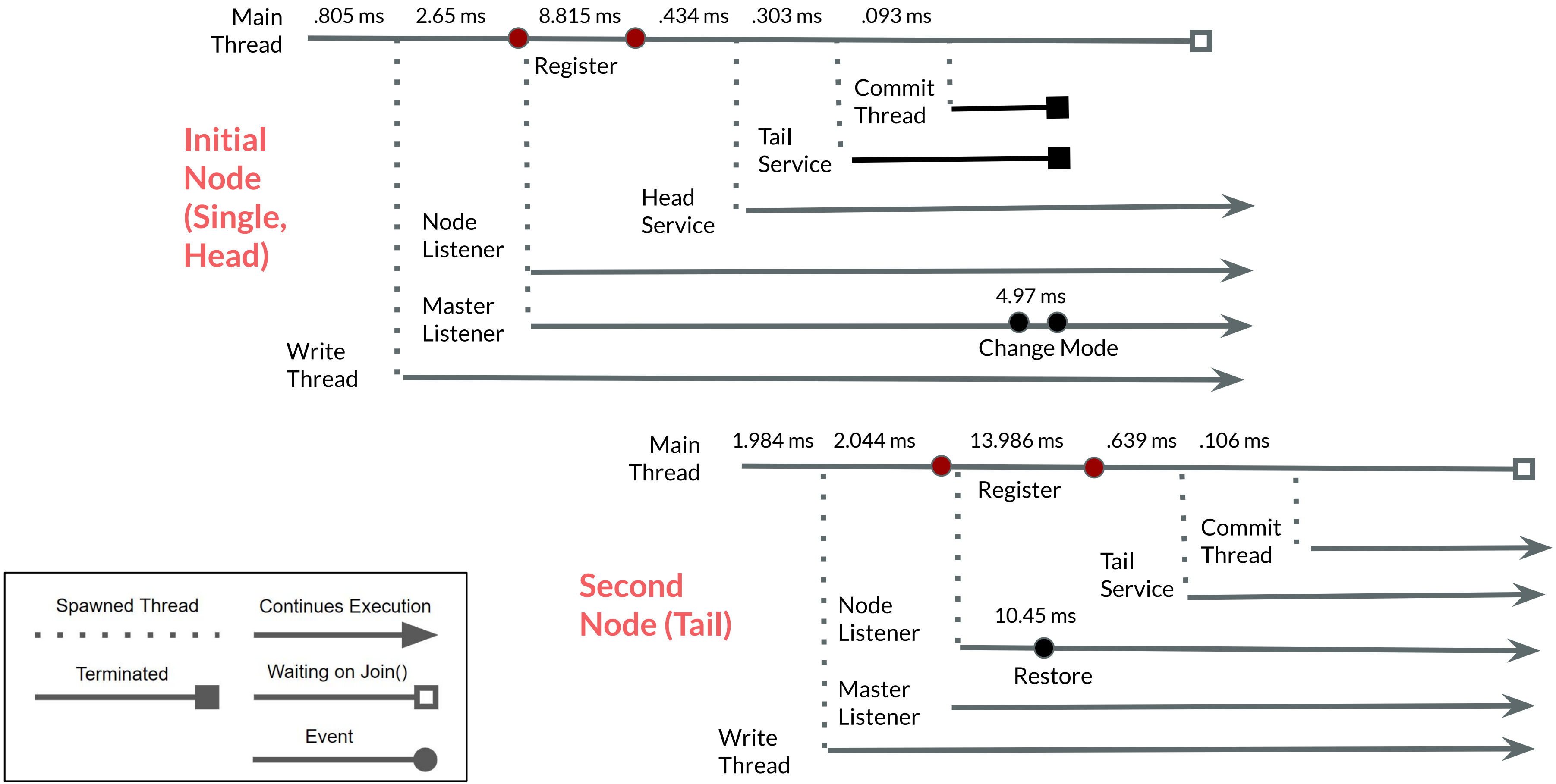


Registration & Restoration

1) At start of restoration, tail pauses writes and write ack relays
2) Tail uses last seq# from new node to get all writes past this point from the volume, along with a copy of the replay log and sent list
3) Tail sends data from each of these to the new node bringing its state in line with the chain, and then resumes normal operations


Master
ChangeMode(Last Seq)
Register (last seq)
2
V
3
Tail
New
Replay Log
~~~
Sent List
~~~
1
Tail
w_r
Sent List
a_r

Integration Timeline



Initial Node (Single, Head)

Main Thread

.805 ms

2.65 ms

8.815 ms

.434 ms

.303 ms

.093 ms

Register

Commit Thread

Tail Service

Head Service

Node Listener

Master Listener

4.97 ms

Change Mode

Write Thread

Second Node (Tail)

Main Thread

1.984 ms

2.044 ms

13.986 ms

.639 ms

.106 ms

Register

Commit Thread

Tail Service

Node Listener

10.45 ms

Restore

Master Listener

Write Thread

Spawned Thread

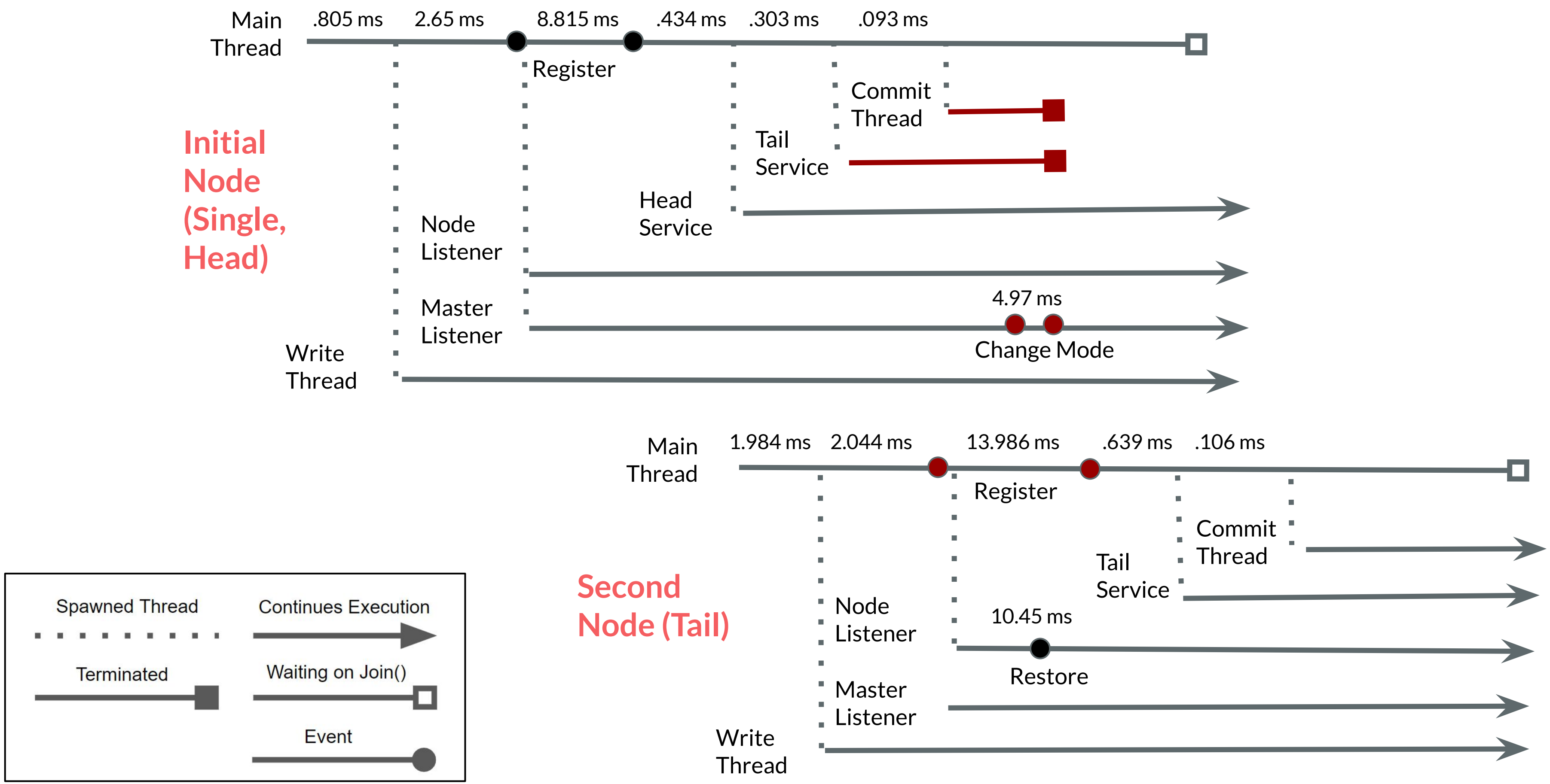
Continues Execution

Terminated

Waiting on Join()

Event

Integration Timeline



Initial Node (Single, Head)

Main Thread

.805 ms

2.65 ms

8.815 ms

.434 ms

.303 ms

.093 ms

Register

Commit Thread

Tail Service

Head Service

Node Listener

Master Listener

4.97 ms

Change Mode

Write Thread

Second Node (Tail)

Main Thread

1.984 ms

2.044 ms

13.986 ms

.639 ms

.106 ms

Register

Commit Thread

Tail Service

10.45 ms

Node Listener

Restore

Master Listener

Write Thread

Spawned Thread

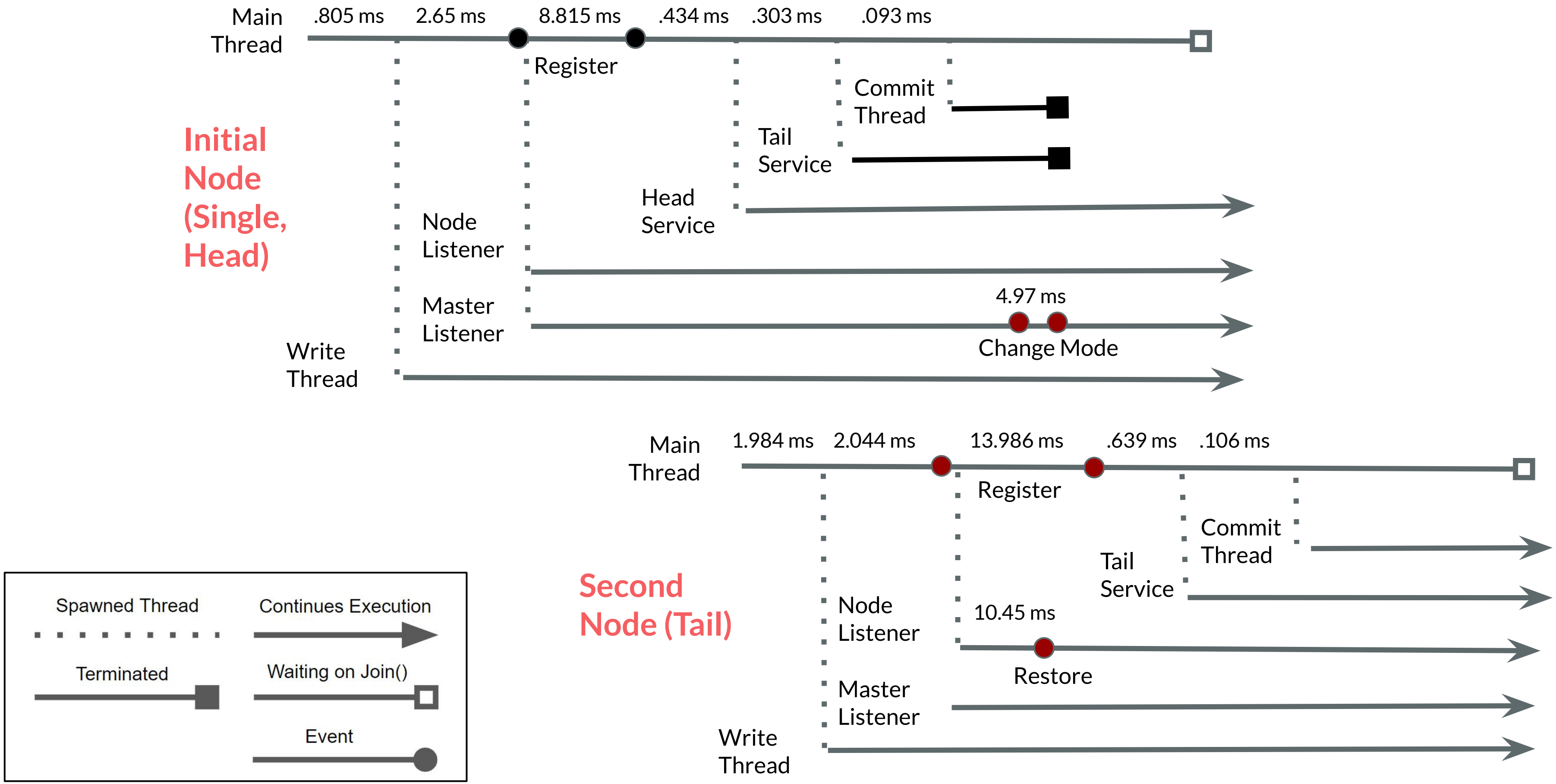
Continues Execution

Terminated

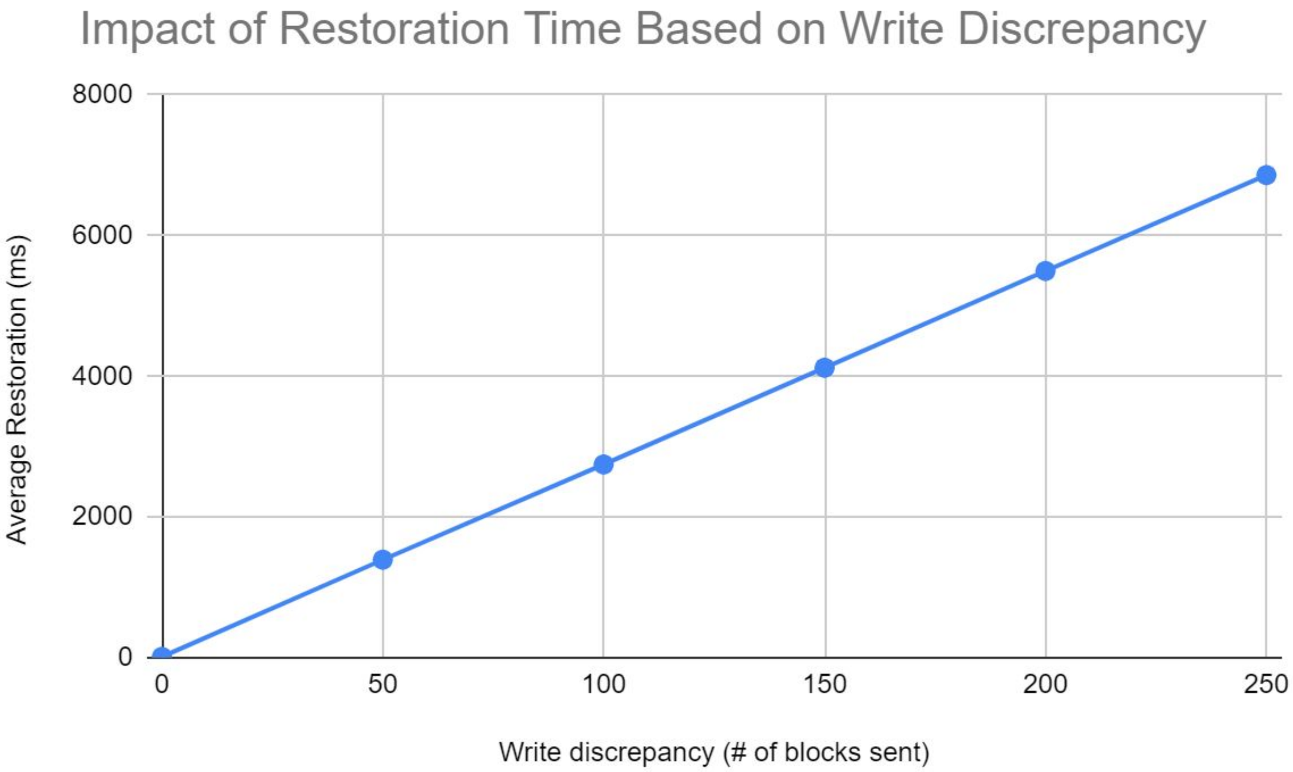
Waiting on Join()

Event

Integration Timeline



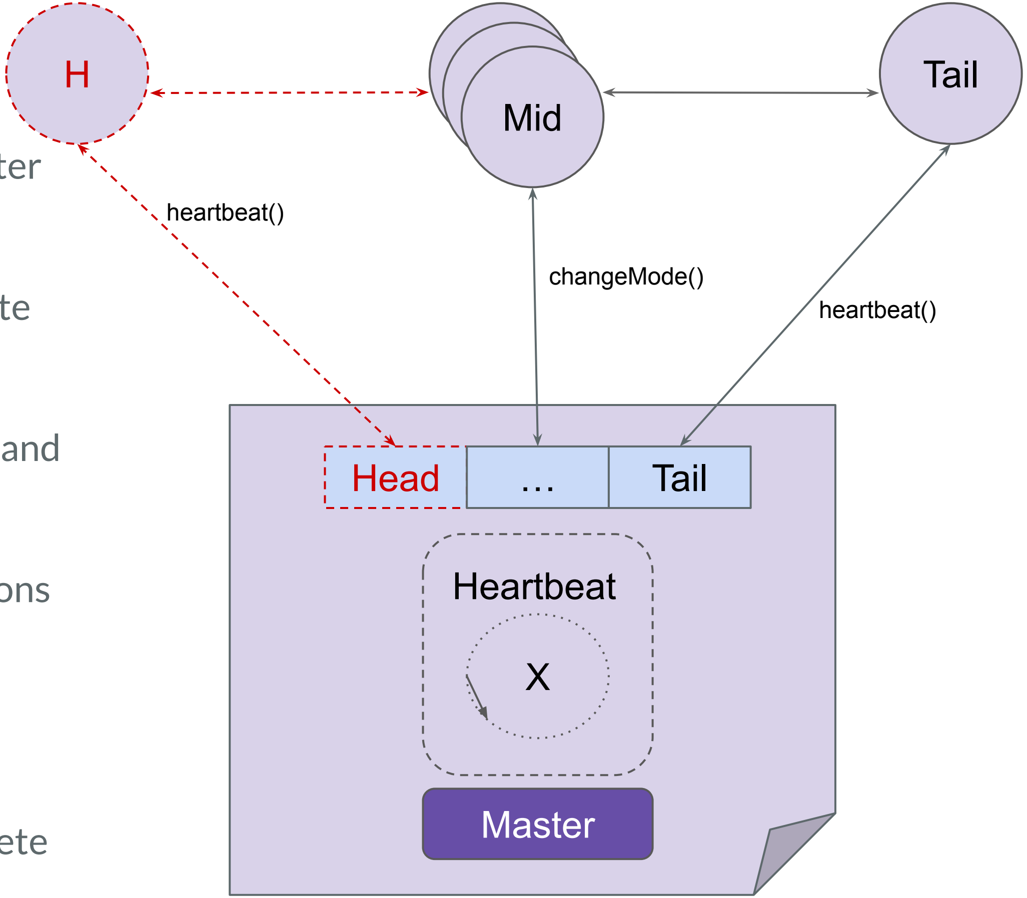
Integration Timeline



Heartbeat

- Thread wakes up and iterates through master node list sending heartbeats to each server
- On detection of failure master will
  - Identify replacement nodes and update virtual node states
  - Removes failed nodes from node list
  - Informs impacted nodes of new state and changes in up\downstream ip’s
  - Update head\tail pointers as needed
- Impacted nodes make necessary modifications to recover including updating local state
- Thread goes to sleep for 5 seconds

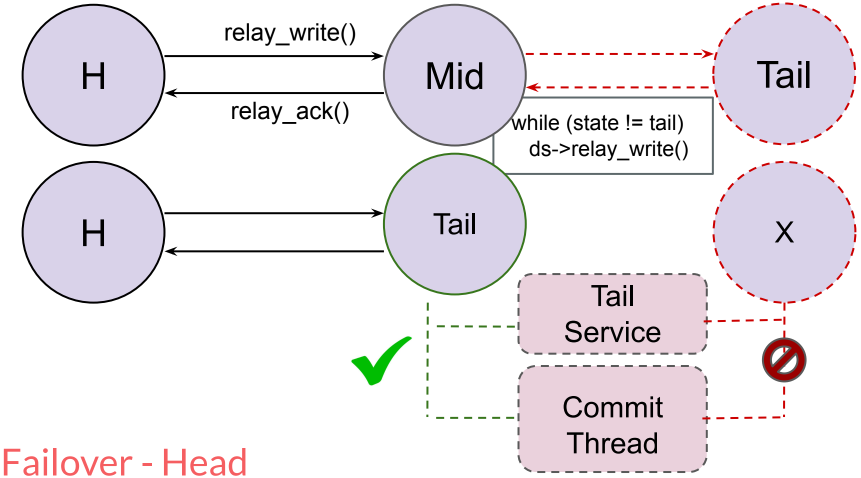
Can deal with multiple failures at once and complete chain failures



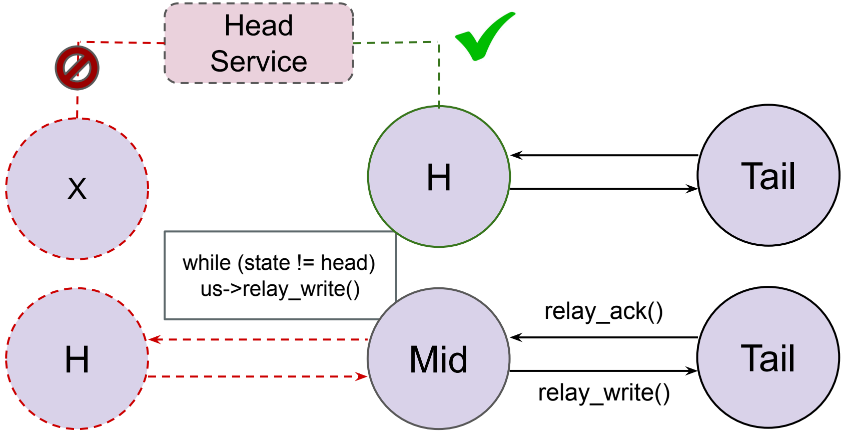
Failover - Head\Tail

- Simple case - Head and tail failover follow similar mechanisms and have minimal impact.
- Both write and commit threads attempt to send messages while wrapped in state checks. If adjacent node fails will continue to try and send until state is updated
- New endpoint will launch relative endpoint services
- For tail failure, new tail launches relay_write_ack thread which starts committing sent

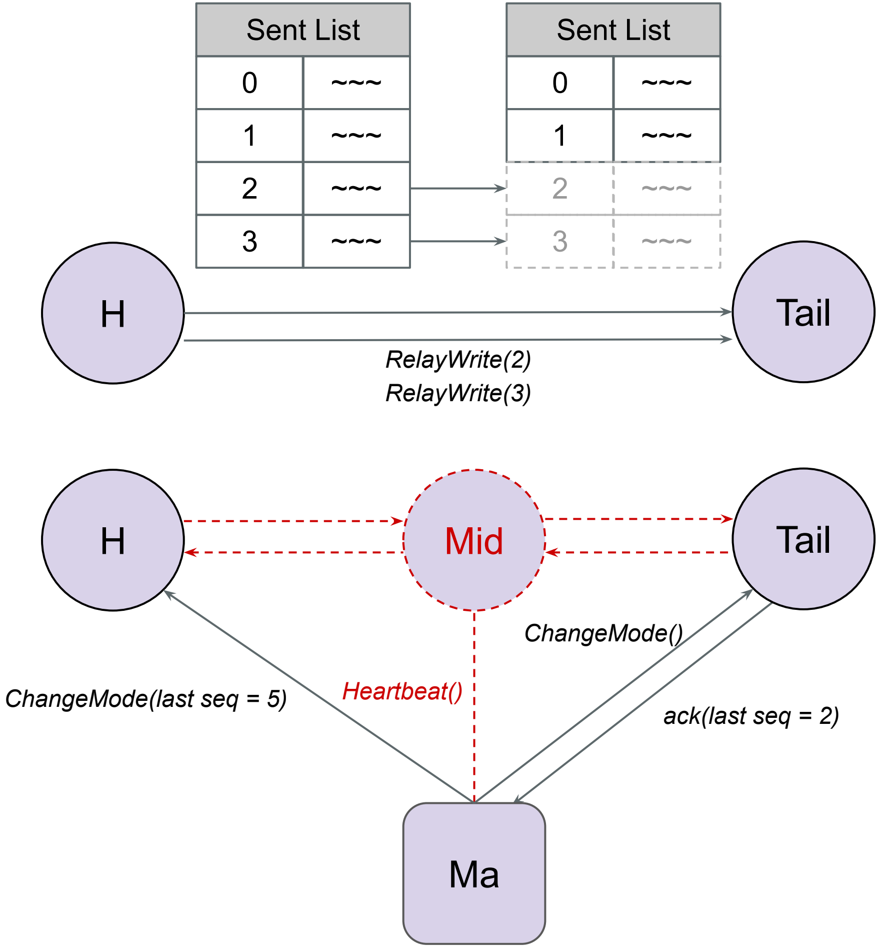
Failover - Tail



Failover - Head



Failover - Mid



Sent List	
0	~~~
1	~~~
2	~~~
3	~~~

Sent List	
0	~~~
1	~~~
2	~~~
3	~~~

- Mid failure involves internode communication to resolve
- M+1 is notified of change by master, and responds with highest sequence number in sent list
- Master notifies M-1 of change, and send along highest sequence seen by M+1
- M-1 resends writes to M+1 that were lost during the failure

Consistency

- Crash on command in code
    - ‘Crash offset’ code consisting of target IP and crash point
    - Server code decodes offset, crashes if IP/crash point match, passes on as offset 0 if not
    - Trigger different crashes at specific intermediate servers and arbitrary points by modifying client request, not server code
- Checksums: verify volume checksums along the entire chain
- Demos
    - Head failure
    - Mid failure
    - Tail failure
    - Re-integrating failed head as new tail

Videos: https://drive.google.com/drive/folders/1y1STB9Z9zsc5_6quiDZosHFA7gRL3UF3?usp=sharing

Head Failure

Mid Failure

Tail Failure

Re-Integration

FIN